Supporting Dynamic Crosscutting with Partial Behavioral Reflection: a Case Study

Leonardo Rodríguez[1] Éric Tanter[2,3] Jacques Noyé[4,3]

[1] Universidad de la República, Instituto de Computación
Julio Herrera y Reissig 565, Montevideo, Uruguay
[2] Universidad de Chile, Departamento de Ciencias de la Computación
Av. Blanco Encalada 2120, Santiago, Chile
[3] OBASCO Project, Ecole des Mines de Nantes – INRIA
4 rue Alfred Kastler, Nantes, France
[4] INRIA Rennes, Campus universitaire de Beaulieu, Rennes, France
lrodrigu@fing.edu.uy etanter@dcc.uchile.cl noye@emn.fr

Abstract

The relationship between reflection and aspect-oriented programming is still under exploration. This paper reports on an experiment to support a widely-accepted, general-purpose aspect language mechanism –AspectJ's dynamic crosscutting–, with a model of partial behavioral reflection. We present a first approach to such a mapping, identify some extensions that can improve the effectiveness of the mapping, and validate our proposal through a revisited mapping. These extensions have been integrated into our Java reflective platform, Reflex.

1. Introduction

Since the advent of object-oriented programming (OOP), a number of advanced modularization issues have been pointed out, for which OOP (as well as other programming paradigms) does not suffice. In particular, the notion of properties that *crosscut* the natural hierarchical decomposition of a system has been identified as responsible for bad modularization such as *duplicated code* and *code tangling* [16]. Tangled code refers to a piece of code that no more reflects a single concern of the design phase, but rather represents a hardly-intelligible mix of several concerns. Both phenomena significantly reduce reuse, evolution and maintenance perspectives.

The last decade has seen the emergence of a great variety of models and proposals for advanced *separation of concerns* [8, 26, 2]. In particular, much attention has been paid to the benefits of using reflection and metaprogramming [20, 38, 7, 37]. Since reflection supports the notion of *metacomputations*, that is, computations about computations, it provides a powerful framework for dealing separately with different concerns. It does so by separating *base* computation and metacomputations into two different levels: the *base level* and the *metalevel*. These levels are causally connected [19]. This means that, on the one hand, a reflective program running at the base level has access to its representation at the metalevel, and that, on the other hand, a modification of this representation will affect further base computations. Various applications of reflection and metaprogramming techniques have been made to achieve better separation of concerns, typically by implementing technical, crosscutting concerns such as distribution, parallelism, fault-tolerance, and synchronization at the metalevel [9, 30, 25, 36, 22].

Experiments with reflection in the context of object-oriented programming has led to the study of *metaobject protocols* [14, 13, 5] and *open implementations*, showing that an object-oriented organization of the metalevel brings various benefits in terms of extensibility and modularity. In 1992, Gregor Kiczales and his group at Xerox PARC began to identify a new model of abstraction in software engineering [12], which finally led to the introduction of *aspect-oriented programming* (AOP) in 1997 [16]. Since then, a significant activity around AOP and related modularization technologies has taken place, as illustrated by the regular tracks on AOP in language conferences such as ECOOP and OOPSLA, and the success of the ACM AOSD conference, dedicated to issues in aspect-oriented software development. Among the great variety of AOP proposals, AspectJ [15] is a reference: it is a simple, well-designed and production-quality extension to the Java programming language, which allows a modular implementation of crosscutting concerns. The AspectJ language is a general-purpose aspect language, as opposed to domain-specific aspect languages [29, 3, 18, 24].

Aspect-oriented programming is deeply connected to reflection and metaprogramming. In short, AOP can be seen as a principled, structured, and language-supported way of doing meta- and reflective programming. The present paper lies in the context of such a connection between AOP and reflection. We have been working on metaobject protocols for behavioral reflection (i.e., reflection about the *behavior* of an application), introducing a fine-grained model for partial behavioral reflection [34]. Recent extensions of this work draws from the observation that there are many AOP proposals out there and much more to explore, which have to "reinvent the wheel" each time, and furthermore are not compatible between each other. We aim at the analysis and design of a reflective kernel for AOP, which can be used to effectively explore the design space of AOP and support new AOP features and languages. A major dimension of such an AOP kernel is to provide support for the composition and collaboration between approaches [33, 32], hence being a major factor of consolidation in the field.

In order to validate our claim that partial reflection is an appropriate low-level framework for supporting AOP approaches in an open manner, we need to carry out consequent case studies of partial behavioral reflection. We have already studied the support of a lightweight domain-specific aspect language for concurrent programming [4, 33], which gave us confidence in the direction we are taking. However, to seriously validate our approach, we need to study the support of a general-purpose, expressive, and widely-accepted aspect language, e.g., the AspectJ language. This paper reports on the first and most fundamental step: studying the mapping of AspectJ's dynamic crosscutting mechanism.

The structure of this paper is as follows. Section 2 briefly introduces partial behavioral reflection and Reflex, our Java open implementation. Section 3 gives an overview of AspectJ's dynamic crosscutting mechanism. Once necessary concepts have been introduced, Section 4 discusses a preliminary approach to supporting AspectJ, from which we identify two extensions to our model of partial behavioral reflection. Such extensions are not necessary to handle AspectJ, but are needed in order to achieve efficiency and elegance. Section 5 details the mapping with the two extensions at hand. Finally, Section 6 discusses related work and Section 7 concludes with perspectives.

2. Partial Behavioral Reflection

Partial behavioral reflection [34] is a model of behavioral reflection that relies on high selectivity, to enhance performance, and configurability, to enhance usability. Our basic model structures the metalevel in terms of *metaobjects* reasoning and acting upon *reifications* of the base-level computation described in terms of *operations* [23]. By operation we mean the basic mechanisms offered by the language, such as message sending, object creation, field access, etc. The *reification* of an operation occurrence is an object representing such an operation occurrence, which metaobjects can therefore manipulate to change the semantics of the operation occurrence.

The combination of reification and metalevel computation is a powerful mechanism, but fairly costly when realized at runtime. Jumping to the metalevel consists of first reifying the operation occurrence, and then delegating (at least part of) its interpretation to the metaobject. In the following, a *hook* is the base-level piece of code responsible for performing a reification and giving control to the associated metaobject(s).

Partial behavioral reflection addresses the issue of flexibility vs. efficiency by limiting, to the greatest extent possible, the number of control flow shifts occurring at runtime. This relies on both *spatial* and *temporal* selection of reification, precisely selecting which entities (classes, objects) are subject to which reifications, and when these interceptions are active.

Our model relies on the notion of *hooksets*, as composable sets that gather execution points scattered in various objects (Fig. 1). Hooksets make it possible to apply a metaobject that modularly implements a concern that *crosscuts* the object decomposition. In our model the metalink –to which we will refer as *link*– which binds a hookset to a metaobject, is made explicit and is characterized by several attributes, such as its scope, control and activation. Scope is used to tailor the granularity of metaobjects (per object, per class, or global). Control makes it possible to specify the time when a metaobject is given control (before, after, or instead of). Finally, the link has an activation condition, which is dynamically evaluated and can be set with different scopes, in order to achieve expressive temporal selection.

Reflex is a portable Java implementation of this model, operating on bytecode. In Reflex, spatial selection is done by specifying *class selectors* and *operation selectors*, predicates that select the points in program text that should be hooked. A hookset in Reflex is either a *primitive hookset*, which consists of an operation class, a class selector and an operation selector, or a *composite hookset*, which is made up of other hooksets, that may be related to various operations. In Reflex, the link is reified as a highly configurable

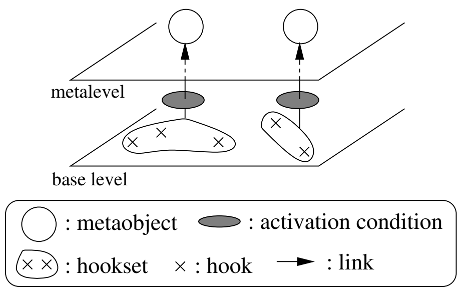

Figure 1. The model of hooksets.

`Link` object that binds a hookset to a metaobject. Furthermore, Reflex is designed as an *open* reflective extension of Java. It is open in the sense that it does not impose any specific MetaObject Protocol (MOP) [14, 13, 5], thanks to a layered architecture. Indeed, Reflex allows metalevel architects to define their own MOP, based on the *Core Reflex* framework, possibly reusing parts of a *standard MOP* library. Architects can define which operations can be reified, and how, by defining an *operation support*. For more details on partial behavioral reflection and Reflex, we refer the reader to [34].

3. AspectJ

AspectJ [15] extends the Java language with a new unit of modularity, *aspects*, to implement crosscutting concerns modularly. AspectJ supports two kinds of crosscutting: *dynamic crosscutting* makes it possible to define additional behavior to run at certain well-defined points in the execution of a program; *static crosscutting* makes it possible to modify the static structure of a program (*e.g.* adding new methods, implementing new interfaces, modifying the class hierarchy)[1]. This study is concerned with the most distinguishing mechanism of AspectJ: dynamic crosscutting. AspectJ follows the Pointcut and Advice model for AOP [21, 35].

3.1. AspectJ basics

In AspectJ a join point represents a well-defined point in the execution of a program, where program behavior can be extended with a crosscutting behavior. AspectJ supports different kinds of join points, which correspond to different operations of the underlying language, Java: method call, field set, handler execution, etc. Since a join point is an execution point, it has a static counterpart at the code level, called the join point *shadow* [21]. A join point may further be discriminated by a dynamically-evaluated condition, called the join point *residue* [11], in order to determine whether a runtime occurrence of the join point shadow actually is an expected join point.

The AspectJ language provides the means to group join points of interest into a *pointcut*, in order to specify the places where an aspect actually affects a base application. A pointcut definition may also specify the *context information* that should be passed to the aspect (*e.g.* the arguments of the current join point). Pointcuts are specified using several primitive *pointcut designators* (PCDs) which can be combined using the standard logic operators. For instance, the following AspectJ code:

```
pointcut move(int x, int y):
    call(* Point.moveXY(int,int))
    && args(x,y);
```

defines a pointcut named `move` that combines two primitive PCDs in order to select all calls to method `moveXY` of class `Point`, and expose both method parameters as context information.

Finally, the crosscutting behavior that should be applied upon occurrences of join points matched by a given pointcut definition is called an *advice*. An advice is a method-like construction that defines the additional behavior to execute at certain join points. When defining an advice, one must explicitly bind it to a pointcut. There are five *kinds* of advice, which differentiate the moment at which the advice is executed with respect to the join point execution: before (before the join point execution), after (after the join point execution), after throwing (after the join point execution, returning with an exception), after returning (after the join point execution, returning normally), around (replace the join point execution). An around advice can include a special `proceed` statement to trigger the execution of the join point it replaces. Advices may have parameters, in which case they must be bound to the pointcut context exposure parameters. For instance, the following AspectJ advice:

```
void around(int x,int y): move(x,y){
  proceed(max(0,x), max(0,y));
}
```

simply ensures that a point cannot be moved to negative values of `x` or `y`.

3.2. Properties of PCDs

Pointcut designators in AspectJ do not all have the same properties. They can be characterized based on the following properties[2]:

- *statically matched*: some PCDs can be resolved completely by looking at the program text. Such PCDs may express:

1 The terminology of static and dynamic crosscutting was introduced in [15]; we could alternatively use the terms structural and behavioral crosscutting.

2 A more exhaustive presentation of pointcut designators can be found in [15] and on the AspectJ website [1].

– a *kind restriction*: match only certain kinds of join point, for instance `call`, which matches method or constructor calls (caller side), or `execution`, which matches effective method or constructor executions (callee side).

– a *signature restriction*: restrict join points based on their signatures. AspectJ offers wildcarding in signatures for convenience; for instance, `call(* *.move())` restrict join points of kind `call` to calls to a no-arg `move` method.

– *location restriction*: restrict join points based on the location in source code where they occur. Examples are `within` and `withincode`.

- *dynamically matched*: such PCDs require runtime information to determine whether they match a candidate join point or not. Such checks are implemented by dynamically-evaluated conditions, called *residues*. AspectJ supports three types of residues: *control flow* residues for expressing control-flow based crosscutting, *instanceOf* residues for runtime type checking, and *if* residues for evaluating arbitrary (though `static`) boolean expressions.
- *context exposure*: such PCDs expose join point information to the context. Examples are `arg` and `this`.

4. Mapping Dynamic Crosscutting

In this section, we present how partial behavioral reflection can support the dynamic crosscutting mechanism of AspectJ. The presentation is informal, example-based, and does not enter into details. It just explains how such a mapping is achieved, in order to make clear some limitations of the current model. We end this section by presenting two minor extensions to the current model of partial behavioral reflection, which lead us to a much better mapping, presented in more details in Section 5.

4.1. Scope of the mapping

We hereby only focus on the dynamic crosscutting mechanism of AspectJ. In order to keep the argumentation clear and concise, static crosscutting is not considered. Concerning dynamic crosscutting, we limit ourselves to the main concepts –join points, pointcuts and advices–, and do not address more advanced features such as aspect instantiation, privileged aspects and so on.

4.2. Running example

The mapping is presented using a simple *shape editor system* (Fig. 2). The system manages three kinds of shapes: `Line`, `Point` and `Composite`. A `Line` has two edges points. `Composite` is a shape container. All of them are subclasses of `Shape`, which is an abstract class with one method, `moveXY`. This method is meant to move the center of a concrete shape to the specified coordinates, therefore, for `Point` it moves the point to the new coordinates, for `Line` it moves the middle point of the line to the new coordinates, consequently moving its edges, and for `Composite` it move the center of the overall shape to the new coordinates, consequently moving all its inner shapes.

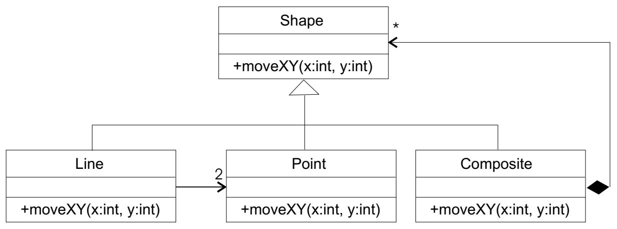

Figure 2. Simple shape editor system

4.3. Initial approach

In this section we present an overview of the mapping of AspectJ dynamic crosscutting to the model of partial behavioral reflection as presented in [34]. Conceptually, an aspect is a metalevel entity, since its primary subject matter is to affect the execution semantics of another program.

A first possible mapping of AspectJ on top of behavioral reflection would consist in reifying whatever occurs in the base program (similarly to a debugging mode) and pass it to some “aspect controller” (a global metaobject) that will dynamically determine which pointcuts match, and execute the corresponding advices. Needless to say, such an approach would be far from satisfying in terms of efficiency.

Rather than entirely matching pointcuts dynamically, partial behavioral reflection gives room for a more staged approach whereby the static projections of pointcuts (their shadows) are represented as hooksets. Hence only execution points of potential interest are reified. Links are configured to delegate control to the appropriate metaobject when needed. The metaobject is then in charge of completing pointcut matching and, if appropriate, executing the advice. This section gives more details on this approach, evaluates it, and concludes with a suggestion of how to improve the model of partial behavioral reflection presented in [34].

4.3.1. Pointcuts As seen in Section 3.2, pointcut designators may impose statically and dynamically matched restrictions over join points. The main matter of pointcut mapping lies in how such restrictions are expressed within Reflex.

Let us first consider static restrictions, with the following simple user-defined pointcut designator, `move`:

```
pointcut move(): call(* *.moveXY(int, int));
```

This pointcut designator uses AspectJ wildcard mechanism: it refers to any invocation of a method `moveXY(int,int)` occurring in any class. In Reflex, mapping the `move` PCD is done simply by defining a primitive hookset characterized by:

- the `MsgSend` operation class;
- a class selector that selects all classes;
- an operation selector that selects only message send occurrences for a method named `moveXY` that receives two `int` arguments.

Note that the `call` PCD can entirely be matched statically, based on program text. In other words, it is fully determined by its shadow. In Reflex vocabulary, a join point shadow is a hook. By extension, a pointcut shadow is a hookset. Hence, mapping PCDs that are statically determined is straightforward in Reflex.

Now consider the following extension to the previous pointcut designator:

```
movePoint(): call(* *.moveXY(int, int))
             && target(Point);
```

This pointcut adds the restriction that the target object of a call to `moveXY` be of type `Point`. Such a restriction cannot be completely resolved statically (if it can) without using an expensive static analysis of concrete types. In program text, it is only possible to select calls that are done on an object whose declared type is either `Point`, or any super and subclass of it. Since the concrete type of a variable is only determined at runtime, this pointcut requires a residue that dynamically checks the type of the target of the call (via `instanceof`). In Reflex, residues do not have a direct counterpart[3]. However, they can be handled as an extra condition checked at the metalevel. When a reification occurs, the context information is all passed to the metaobject controlling the considered hookset. Such a metaobject starts by checking the residue condition before continuing.

Residues are all checked in the same way. However, a control flow residue deserves special attention. The pointcut designators related to control flow (`cflow` and `cflowbelow`) allow picking out join points based on whether they are in a particular control-flow relationship with other join points. Checking this relationship represents nothing new compared to other residues, however, setting up this relationship implies exposing the control flow information of other join points.

Let us discuss this with an example:

```
moveSinglePoint(): movePoint()
                   && !cflowbelow(move());
```

3 Only activation conditions are available, which are not meant to reason about context information other than the currently executing object [34].

This PCD further restricts the `movePoint` PCD, matching only those invocations that are not made below the control flow of any join point matched by the `move` PCD. In other words, it matches only the moving of a stand-alone point (top-level calls). Note that this pointcut definition implies the definition of two nested pointcuts: the one inside the `cflowbelow` (`move`) and the one affected by a control flow restriction (`movePoint`).

To be able to determine whether `movePoint` is matched in the control flow of `move`, we first need to expose the control flow information of `move`: this is done using the notion of *event collectors* [33]. Event collectors gather execution events to expose parts of a program execution (nesting, sequences, etc.), under any structure (counter, stack, tree, DAGs, graphs, etc.), for dynamic introspection. In particular they are used to expose control flow information. Indeed, event collectors are just like metaobjects, except that their purpose is only to *expose* elements of program execution, rather than to *affect* program execution. In order to support `cflow` and `cflowbelow`, exposing a simple counter that keeps track of entries and exits in a pointcut suffices. Consequently, mapping a control flow PCD implies two separate tasks:

- defining a separate link for an event collector exposing control flow information of the pointcut passed as argument to `cflowbelow`;
- defining a condition that checks the exposed control flow information.

Hence, in our example, the metaobject must check two residues: the control flow condition, and the instanceof condition.

4.3.2. Pointcut parameters Consider the following extension of the pointcut designator `moveSinglePoint`:

```
moveSinglePointArg(int x, int y):
  moveSinglePoint() && args(x,y);
```

This pointcut designator extends the previous example by exposing the arguments of the invocation of method `moveXY(int,int)` in `Point`. These arguments will then be available when defining an advice bound to this pointcut.

In Reflex, the information that is reified is defined at the operation level: when support for a particular operation (*e.g.* message sending) is added to Reflex, an entity responsible for the low-level code transformation is given. Such an entity (called a hook installer [34]) determines which information is reified and how. This unfortunate coupling implies that we either need to define a new operation support each time a different piece of context information is needed, or always reify all context information and let the metaobject extract the needed pieces.

4.3.3. Advices

Since an aspect may contain any legal Java class member, it must be mapped to an ordinary class, following the singleton pattern. It could be mapped to a metaobject class, however, due to the extra work we leave in metaobjects (checking residues, extracting context information), it is preferable to get a better separation and stick to a clean aspect class.

Advices are transformed into methods of the aspect class. Its name is generated by the translator and its arguments are the same as in the advice definition. For instance:

```
aspect MovingPoint {
  // -- pointcut definitions --

  after(int x, int y):
             moveSinglePointArg(x,y){
    log.println("Point moved to: "
                + x + "," + y);
} }
```

This logging aspect is transformed into a class with the same name, with a method corresponding to the advice (Fig. 3). Besides, the pointcut `moveSinglePointArg` results in one hookset `hs-move` (since the shadow of this pointcut is defined by the pointcut `move`) and the metaobject `moveSPA`. The associated metaobject is responsible for checking the residues (for `cflowbelow` and `target`), collecting the values of the arguments, and finally invoking the aspect advice method. The binding between the metaobject and the hookset is done through a link. The kind of the advice (before, after, around)[4] is mapped to the control attribute of the link (in this case, `after`).

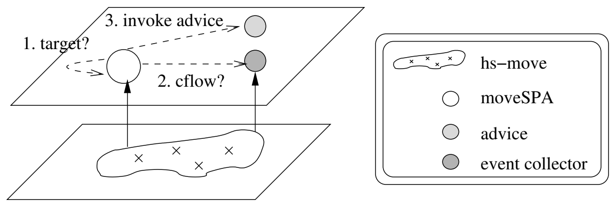

Figure 3. Initial mapping of the aspect `MovingPoint`

Since `moveSinglePointArg` is affected by a control flow restriction over the pointcut `move`, an additional link is required: this link binds the same hookset `hs-move` to an event collector metaobject. Such a metaobject is a simple counter that keeps track of entries and exits in the hookset. Therefore the link control is set to `before_after`.

4 Reflex does not yet support *after throwing*, although there are plans to support it in a near future.

4.4. Issues of the Mapping

Our initial approach shows that the model of partial behavioral reflection as presented in [34] is expressive enough to handle the dynamic crosscutting mechanism of AspectJ in a non-naive manner (conversely to the all-dynamic approach consisting of a centralized pointcut matching process occurring at runtime). However, this experiment also highlights interesting issues that could be resolved to achieve a better mapping.

First of all, pointcuts are not handled as efficiently as they could, since we are not able to embed residues in the code: they need to be checked at the metalevel. This has the bad property of forcing reification even in cases where the residues reject a particular occurrence. And reification is a major source of overhead in reflective systems.

Furthermore, the handling of selective pointcut parameter exposure is not satisfactory: one of the alternatives mentioned in section 4.3.2 implies reifying all available pieces of information, which is highly inefficient, and the other one implies defining new operation supports each time a different piece of information is required. Recall from [34] that an operation support is made up of an operation class and a hook installer. This last alternative is efficient since hook installers determine the required pieces of information, but would be really cumbersome in practice: it might result in a myriad of defined operation supports for what is conceptually the same operation (which just happens to be reified differently).

Finally, these two limitations forced us to introduce an extra indirection from metaobjects to aspect objects, in order to cope with them. In the next section we introduce two extensions to the model of partial behavioral reflection that make it possible to circumvent these limitations. We aim at a mapping that is both efficient and clear. Clarity would be obtained by having metaobjects simply implement advices, embedding other concerns within hooksets and links.

4.5. Two model extensions

In order to solve the issues presented above, we propose two extensions to our model of partial behavioral reflection:

MOP descriptors – to describe, *at the link level* (as opposed to the operation level), which information has to be reified, and how, upon occurrences of a given operation;

hookset restrictions – to embed dynamically-evaluated, but fixed, conditions in generated hook code.

4.5.1. MOP descriptors

A MOP descriptor is an object that describes how an operation should be reified: it defines the expected type of the metaobject as well as the method to invoke. This description is completed by a specification of the parameters that should be passed to the metaobject method. The metaobject class should obviously be compatible with the specified type and implement the method corresponding to the given name and parameters.

Parameters are simply objects that implement a dedicated interface, `Parameter`:

```
interface Parameter {
  public String getCode(Operation aOp);
}
```

The role of a parameter object is to generate the source code that, when executed, results in the reference to the desired information. Standard parameters are provided, such as `CONTEXT`, which refers to the currently executing object (or class if within a static member), or `TARGET`, which refers to the target object of the considered operation occurrence. Another example is a parameter that resolves to the reference to the metaobject associated to a given link. Operation-specific parameters are also provided by hook installers via *parameter pools*. For example, for the `MsgReceive` operation, the parameter pool makes it possible to obtain a parameter object for a method object, or a parameter (at a given index) of the invocation. Finally, custom parameters can be specified. The user then needs to give the source code, in the extended Java language supported by Javassist [6], that should be evaluated.

A MOP descriptor makes it possible to specify whether a set of parameters should be passed as plain arguments to the metaobject, packed into an object array, or encapsulated into some object, instance of a user-defined class.

Since MOP descriptors completely decouple the specification of the MOP from the capability of reifying an operation (implemented in hook installers), it is now possible to use different MOP descriptors for the same operation in different links, which greatly simplifies the mapping of the pointcut context exposure feature of AspectJ.

4.5.2. Hookset restrictions A hookset restriction is a dynamically-evaluated condition that should be true in order for a hook to trigger reification and metaobject invocation. Such a restriction is hardwired in the hook code at generation time to improve performance. Hookset restrictions are specified when creating a link. In case the hookset bound to a given link is compound, it is possible to set a restriction that applies to all or only some of the sub-hooksets.

The restriction is specified as a static method and hence can be computed based on globally-available information (*i.e.* static fields or methods), as well as parameters if needed. Parameters are specified when declaring the restriction in a similar manner as for MOP descriptors. The list of parameters must be compatible with the signature of the restriction method.

For example, pointcut `movePoint` implied a restriction that the target object of a `moveXY` call be of type `Point`. This pointcut introduces the need for an `instanceOf` residue, which can be specified as the following restriction, to which the parameter `TARGET` is given:

```
public static boolean accept(Object o){
  return o instanceof Point;
}
```

When attached to the hookset selecting calls to `moveXY` this restriction discards, at runtime, calls to objects which are not of a subtype of `Point`.

5. Revisiting the Mapping

In this section we revisit the mapping of the dynamic crosscutting mechanism of AspectJ to Reflex, taking into account the two extensions to the model of partial behavioral reflection presented in Section 4.5. This section illustrates that these extensions allow us to greatly improve the quality of the mapping, both in terms of efficiency and clarity. Besides explaining how such extensions do enhance the mapping, we give a more detailed presentation of the involved translation.

5.1. Pointcuts

Thanks to the extensions presented above, we are able to completely map a pointcut declaration without having to postpone extra responsibilities to the metaobject. In other words, the result of translating a pointcut is a (possibly composite) hookset, along with hookset restrictions and appropriate information for the MOP descriptor of the link. We hereby give an overview of the pointcut translation process.

5.1.1. The translation process Once parsed, a pointcut is represented by a tree with PCDs as leaves, and logical operators as nodes. Our translation algorithm is pretty straightforward. The first step consists of building an intermediate isomorphic tree in which each leaf is replaced by a quadruple. Such a quadruple is generated based on the properties of the PCD (see Section 3.2). It is of the form (PH, SR, DR, CE), where:

- PH represents a primitive hookset. PH is non-empty only if the given PCD restricts the join point kind, such as `call`, in which case PH holds the operation that corresponds to the PCD kind. If the PCD is further restricted by a signature pattern, such a pattern is mapped to class and operation selectors.
- SR is a static restriction expression. SR is non-empty only if the given PCD serves for stating location or signature restrictions, such as `within`.
- DR is a dynamic restriction expression that corresponds to the residues of a PCD, if any. Recall that PCDs such as `this` and `cflow` do require residues, whereas `call` does not.

- CE holds the list of context parameters exposed by the PCD, if any.

The second step of the translation algorithm is to reduce the tree as much as possible, before converting it to elements of Reflex. This reduction is done by applying a set of reduction rules that basically discard nodes and compose the quadruples accordingly. The principle followed by the reduction process is simple: eliminate all the nodes with `!` or && operators. To this end, reduction rules are applied in a given order, and most of them are straightforward; for instance, "in case at least one child of an && operator is an `||` operator, perform distributive composition of quadruples", or, "the negation of a quadruple reduces to the negation of all its components".

Distributive composition of quadruples is done as follows: PHs that relate to the same operation are composed together via primitive hookset composition operators (union, intersection, etc.); SRs, as well as DRs, are composed together using the usual logical operators; and CEs are composed together by simply merging their lists. Negating PHs implies negating corresponding class and operation selectors, while negating SRs and DRs simply means negating the expressions (or nothing if empty). Once the reduction rules have been applied the PH is composed with the SR by merging the additional static restriction expression in SR into additional class and/or operation selectors in PH.

If the resulting tree is a single leaf, then it represents a single primitive hookset. Otherwise, the tree represents a composite hookset: all nodes are union operators and leaves contain the different primitive hooksets with their associated dynamic restrictions and context exposure requirements. To finally define the pointcut in Reflex, the resulting (composite) hookset is created and defined. When defining the link, for each sub-hookset, the hookset restrictions are specified and the MOP descriptor is configured to reify the required parameters.

5.1.2. Illustration Let us explain how the pointcut `movePoint`, presented previously, is translated. First, it is represented by a tree with one node (the logical &&) and two leaves: one for the `call` PCD, which embeds its signature restriction and one for the `target` PCD, with its type restriction.

The `call` PCD is replaced by the quadruple *({ph}, {}, {}, {})* where *ph* is the primitive hookset described in Section 4.3.1 for the pointcut `move`. The `target` PCD expresses a restriction that is partially matched statically (the target type must be either `Point` or a subtype or supertype of `Point`). It must be completed by a dynamic restriction (`instanceOf(Point)`). Hence, this PCD is replaced by a quadruple *({}, {sr}, {dr}, {})*, where *sr* and *dr* represent the static and dynamic restrictions respectively. Note that since this PCD does not impose any kind restriction, the PH component of the quadruple is empty.

This tree is simply reduced by composing both quadruples, resulting in a quadruple *({ph}, {sr}, {dr}, {})*. To finish, *sr* is composed with *ph*.

Now, consider the pointcut `moveSinglePoint`, which adds a control flow restriction to `movePoint`. The `cflowbelow` PCD is replaced by a quadruple *({}, {}, {dr'}, {})*, where *dr'* expresses the control flow condition to be dynamically checked. In addition, the translation process triggers the translation of the internal pointcut `move`, in order to generate the required control flow information (via an event collector, recall Section 4.3.1). The reduction implies negating the `cflowbelow` quadruple (thus negating *dr'*) and composing it with the `movePoint` quadruple. The resulting dynamic restriction would then be *dr && !dr'*, which would be implemented by the following hookset restriction:

```
public static boolean accept(Shape s){
  return (s instanceof Point) &&
         !(MoveCflow.getCounter() > 1);
}
```

5.1.3. Evaluation Using hookset restrictions and MOP descriptors, we do not need anymore intermediate metaobjects between the cut and the aspect behavior: metaobjects can simply be objects that implement advices.

Indeed, using hookset restrictions to implement residues avoids the need for checking residues at the intermediate metaobject level. Using MOP descriptors to specify which information should be reified frees the intermediate metaobject of the task of collecting parameters before calling the associated behavior. Furthermore, since a MOP descriptor also specifies which method to call on the metaobject, we can directly invoke the appropriate advice method.

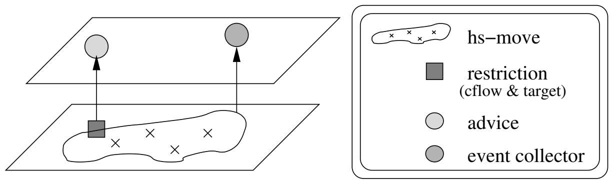

Figure 4. Revised mapping of the aspect `MovingPoint`

Hence, the two extensions to the model of partial behavioral reflection not only enhance performance significantly, they also greatly simplify the overall picture of the mapping: pointcuts are mapped to hooksets and restrictions, advices to metaobject methods, and binding specification is handled in the link (context exposure, advice kind, etc.). Fig. 4 shows the revised mapping of the aspect `MovingPoint` (to be compared with Fig. 3.

5.2. Advices

We are now able to simply map an advice to a metaobject method that receives the advice parameters. Apart from this simplification, the mapping principle remains the same as exposed in the previous section. Nevertheless, we now discuss some advanced issues with advices that we ignored in the previous section. Advice bodies in AspectJ have two special features that make them different from simple Java methods: `proceed` statements, and access to reflective information about a join point.

5.2.1. Proceed statement An around advice traps the execution of a join point and runs instead of it. The original computation of the join point can be invoked through a special `proceed` statement, which acts as a method call. It accepts the same exposed parameters of the original join point, and returns what the original join point returns. The arguments of the `proceed` may actually replace the original values (Section 3).

In Reflex, a standard supported operation has a corresponding *dynamic operation* class [34]. A dynamic operation encapsulates runtime information describing an operation occurrence, and also provides a `perform` method that actually executes the intercepted operation occurrence.

As seen in Section 5.1.1, MOP descriptors can be configured to specify the parameters required by each quadruple. In case of an around advice, we specify an extra implicit parameter which is a command object [10] wrapping a dynamic operation. The command object embeds operation-specific logic related to mapping `proceed` parameters to that of the dynamic operation. All `proceed` statements in an advice body are subsequently replaced by execution of the command object.

5.2.2. Join point reflective information AspectJ provides three implicit references that provide reflective information of the current join point. They can be used from the body of any advice or from an `if` PCD. These three implicit references are:

- `thisJoinPoint`: provides reflective information about the static part (*i.e.* the signature of the join point) and the dynamic part (*e.g.* the values of the arguments) of the current join point.
- `thisJoinPointStaticPart` is a subset of the `thisJoinPoint`, which only provides reflective information about the static part.
- `thisEnclosingJoinPointStaticPart` provides reflective information about the static part of the *enclosing* join point, that is to say, the method (or constructor, or static initializer) in which the current join point occurred.

These instances essentially collect all the current join point context information, plus the reflective static information describing the actual signature of the join point and expose it by implementing a set of interfaces defined by AspectJ.

As seen in Section 4.5, parameters specified in MOP descriptors give access to context information of a hook, thus easily handling the dynamic part of `thisJoinPoint`. In addition to this, when parameters are resolved at generation time, they have access to the static information of operation occurrences, a direct counterpart of `thisJoinPointStaticPart`. Furthermore, this information includes information about where the occurrence is located, thereby making it possible to compute `thisEnclosingJoinPointStaticPart`.

Therefore, providing access to these variables in the context of the mapping simply implies providing (1) a set of classes that implement the interfaces defined by AspectJ, and (2) three parameters (one for each instance) that instantiate such classes with the necessary information. These parameters are passed as implicit parameters to the advice body, exactly as in done for AspectJ as described in [11], without having to transform advice bodies.

6. Related Work

The idea that reflection can support AOP is not new (see for instance [31, 17]) and has been a matter of (often mundane) discussion since the beginning of AOP. We have been going one step further by actually showing that an essential part of an efficient AOP language, AspectJ, implemented without resorting to reflection, could actually be supported by reflection in an effective way.

A basis of our proposal has been the work on the implementation of AspectJ [11], which suggested that an efficient implementation of residues was a key issue. We have been able to do this without leaving the framework of reflection.

It is also interesting to compare our work with other proposals that focus on flexible solutions for AOP such as JAC [27] and PROSE [28]. These proposals rely on basic technology that is fairly close to reflection and makes it possible to weave aspects at runtime. However, their implementation is sealed and only accessible via a predefined high-level language. Also, dynamic weaving results in a significant overhead due to the use of wrappers in JAC and the Java debugging interface in PROSE. The use of partial reflection and a careful design makes it possible to reduce this overhead without sacrificing flexibility.

7. Conclusion

Not surprisingly, partial behavioral reflection is *conceptually* expressive enough to support the kind of dynamic crosscutting offered by AspectJ. This is not surprising because, conceptually, any form of behavioral reflection can cover dynamic crosscutting. The real challenge is to do so practically, in a natural and direct manner. Partial behavioral reflection as presented in [34] was already a progress in this sense over more classical approaches to behavioral reflection. Still, from the critical study of a first experiment, we have identified and validated two small extensions to our model (hookset restrictions and MOP descriptors) that allow for a clearer and more effective mapping. We are presently carrying benchmarks to validate the interest of our approach.

This work is part of a much larger project around the study of how partial reflection can support AOP mechanisms in a general, extensible, yet efficient manner. This project targets the design and implementation of a versatile kernel for AOP [32, 33]. In this direction, with respect to AspectJ, the next step is to study the support of the other mechanisms, such as static crosscutting.

References

[1] The AspectJ website, 2002. http://www.eclipse.org/aspectj.

[2] J. Brichau, M. Glandrup, S. Clarke, and L. Bergmans. Advanced separation of concerns workshop report. In *ECOOP 2001 Workshop Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[3] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 110–127, Pittsburgh, PA, USA, Oct. 2002. Springer-Verlag.

[4] D. Caromel, L. Mateu, and É. Tanter. Sequential object monitors. In M. Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 316–340, Oslo, Norway, June 2004. Springer-Verlag.

[5] S. Chiba. A metaobject protocol for C++. In *Proceedings of the 10th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, pages 285–299, Austin, Texas, USA, Oct. 1995. ACM Press. ACM SIGPLAN Notices, 30(10).

[6] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In F. Pfenning and Y. Smaragdakis, editors, *Proceedings of the 2nd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2003)*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376, Erfurt, Germany, Sept. 2003. Springer-Verlag.

[7] P. Cointe, editor. *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection (Reflection 99)*, volume 1616 of *Lecture Notes in Computer Science*, Saint-Malo, France, July 1999. Springer-Verlag.

[8] E. W. Dijkstra. The structure of THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[9] J.-C. Fabre, V. Nicomette, T. Pérennou, R. J. Stroud, and Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 489–498, Pasadena, CA, USA, June 1995. IEEE Computer Society Press.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.

[11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 26–35, Lancaster, UK, Mar. 2004. ACM Press.

[12] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA 92 Workshop on Reflection and Metalevel Architectures*. Akinori Yonezawa and Brian C. Smith, editors, 1992.

[13] G. Kiczales, J. M. Ashley, L. Rodriguez, A. Vahdat, and D. G. Bobrow. Metaobject protocols: Why we want them and what else they can do. In A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101–118. MIT Press, 1993.

[14] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[16] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al., 1996.

[17] S. Kojarski, K. Lieberherr, D. H. Lorenz, and R. Hirschfeld. Aspectual reflection. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

[18] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[19] P. Maes. *Computional reflection*. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium, 1987.

[20] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, Alghero, Sardinia, Oct. 1988.

[21] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs.

In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[22] H. Masuhara, S. Matsuoka, and A. Yonezawa. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. In *IPSJ SIG Notes*, volume 94-PRG-18, 1994.

[23] J. McAffer. Engineering the meta-level. In G. Kiczales, editor, *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, pages 39–61, San Francisco, CA, USA, Apr. 1996.

[24] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, Feb. 1997.

[25] H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP 94)*, volume 821 of *Lecture Notes in Computer Science*, pages 299–319. Springer-Verlag, July 1994.

[26] D. Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[27] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In Yonezawa and Matsuoka [37], pages 1–24.

[28] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *1st International Conference on Aspect Oriented Software Development (AOSD)*. Springer-Verlag, 2002.

[29] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain-specific aspect languages. In *OOPSLA 2003 Domain-Driven Development Track*, October 2003.

[30] R. J. Stroud and Z. Wu. *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter Using Metaobject Protocols to Satisfy Non-Functional Requirements, pages 31–52. CRC Press, 1996.

[31] G. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10), Oct. 2001.

[32] É. Tanter and J. Noyé. Motivation and requirements for a versatile AOP kernel. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004)*, Berlin, Germany, Sept. 2004.

[33] É. Tanter and J. Noyé. Versatile kernels for aspect-oriented programming. Research Report RR-5275, INRIA, July 2004.

[34] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In R. Crocker and G. L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, Oct. 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[35] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, Sept. 2004.

[36] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In N. Meyrowitz, editor, *Proceedings of the 3rd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 88)*, pages 306–315, San Diego, California, USA, Sept. 1988. ACM Press. ACM SIGPLAN Notices, 23(11).

[37] A. Yonezawa and S. Matsuoka, editors. *Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan, Sept. 2001. Springer-Verlag.

[38] C. Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.